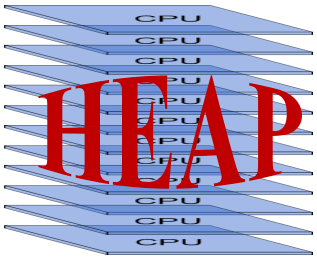
Information and Communication Technologies (ICT) Programme

Project N^{o}: FP7-ICT- 247615

HEAP



Deliverable D3.5

Metric-driven coverage enhancement for profiler

Technical documentation

Author(s): Florian Schaefer (FSResult), Luciano Lavagno (PTO)

Status -Version: V1.1

Date: 5 March 2012

Distribution - Confidentiality: Public

Code: HEAP_D3.5_PTO_V1.1.pdf

Abstract: This document describes the methodology developed for verification in the HEAP project, as well as the annotation API, the analysis tool, and their usage. The purpose is to provide an environment for verifying and assessing the correctness of parallel programs with respect to a sequential reference implementation. The methodology and tools can be applied to both automatically and manually parallelized code. Verification is achieved by first introducing data and control monitoring statements into the sequential and parallel code (this insertion can be easily automated). These annotations produce a trace file when the code is executed, which is then read and used by the analysis tool to check both (1) the source code coverage of the execution, thus measuring confidence in the verification validity, and (2) check the correctness of the data computed by the parallel version at several intermediate checkpoints.

© Copyright by the HEAP Consortium

Disclaimer

This document contains material, which is the copyright of certain HEAP contractors and sub-contractors, and may not be reproduced or copied without permission. All HEAP consortium partners have agreed to the full publication of this document. The commercial use of any information contained in this document may require a license from the proprietor of that information. In particular, this document was prepared by Florian Schaefer (FSResult), a sub-contractor of Politecnico di Torino, and its copyright belongs fully to Florian Schaefer.

The HEAP Consortium consists of the following companies:

No	Participant name	Participant short name	Country	Country
1	ST Microelectronics	STM	Co-ordinator	Italy
2	Synelixis Solutions Ltd	Synelixis	Contractor	Greece
3	Thales Communications	Thales	Contractor	France
4	ACE Associated Compiler Experts B.V.	ACE	Contractor	Netherlands
5	Compaan Design	Compaan	Contractor	Netherlands
6	Politechnico Di Torino	PoliTo	Contractor	Italy
7	ATHENA Industrial Systems Institute	Athena	Contractor	Greece
8	Universita Degli Studi Di Genova	UniGe	Contractor	Italy
9	SingularLogic	SiLo	Contractor	Greece
10	Uppsala University	UPP	Contractor	Sweden

Document Revision History

Date	Issue	Author/Editor/Contributor	Summary of main changes
2012-1-30	0.0	Florian Schaefer	Main contents.
2012-2-25	1.0	Luciano Lavagno	Formatted according to standard.
2012-3-5	1.1	Luciano Lavagno	Responses to comments from Synelixis.

Contents

1 Introduction **3**

2 Quicksort Example **3**

3 Methodology **4**

4 Adaption the Program to the Runtime Environment **6**

5 Including the HEAP-API **6**
5.1 Thread-safe Memory Allocation 8
5.2 External Parallelism Facilities 9

6 Parallelization **9**
6.1 Pool-based Parallelism 10
6.2 Consumer-Producer style Parallelism 13

7 Program Segmentation **16**
7.1 Tasks 18

8 Dumping Data **19**
8.1 Block Data 19
8.2 Stream Data 20

9 Program Annotation **22**
9.1 State 22
9.2 Assertions 23
9.3 Checkpoints 24
9.3.1 Unordered Checkpoints 25
9.3.2 Ordered Checkpoints 26

10 Structure File **27**

11 Compilation & Running **28**

12 Annotation of the Sequential Version **29**

13 Adaption to other Environments **30**

14 HEAPAnalyze **30**
14.1 Options 30
14.2 Assertion Checking 31
14.3 Data Verification 32
14.4 Order Verification 32

© 2012 Florian Schäfer

14.5 Coverage Analysis 33
14.6 Performance Analysis 34

15 The HEAP-API C-library **37**
15.1 Management Functions 37
15.2 Functions for Setting Up External Parallelism 38
15.3 Parallelism 39
15.3.1 Data Ports 40
15.4 Utilities 42
15.5 Reporting 43
15.5.1 Areas and Tasks 43
15.5.2 Data 44
15.5.3 Assertions and Checkpoints 45

16 Appendix **46**
16.1 Quicksort Analysis Output 46

© 2012 Florian Schäfer

1 Introduction

This document describes the methodology developed for verification in the *HEAP* project. Furthermore, the document describes the *HEAP API*, the *HEAPAnalyze* tool and their usage. The purpose of these tools is to provide an environment for verifying and assessing parallel programs.

Verification and assessment are achieved by first introducing tracking statements into the parallel program – either manually or automatically during the parallelization step. The statements are provided by the *HEAP API* library and produce a runfile when the program is run. The runfile then contains all the required information for verification and assessment. This information is finally used by the *HEAPAnalyze* tool to perform the desired verification and assessment steps.

The *HEAP API* also contains parallelism facilities that can be used when the program is parallelized. But this is optional, the tracking statements also work with external parallelism facilities.

Together with the *HEAP API*, a set of program examples has been developed and is distributed together with the software deliverable. , a parallel version of the *quicksort* algorithm, is used throughout the documentation for demonstration purposes and will be introduced in detail in the next section.

2 Quicksort Example

The example program is a parallelized quicksort. The program parses the arguments, initializes the *HEAP API* library, creates random data from a fixed random seed, creates the main task (the task to sort the entire data set) and control is transfered to parallelism facilities to execute the tasks. Finally, the result is verified and the library is finalized. A sequential version also exists. This one is required to create the reference runfile.

The functions of the quicksort example are spread over four files:

- **quick_sort.c** – the functions unique to the parallel quicksort: `main()` and `sort_slave()`.
- **quick_sort_seq.c** – the functions unique to the sequential quicksort: `main()` and `quick_sort_helper()`.
- **shared_quick.c** – the function shared by all quicksort examples (parallel and sequential): `quicksort_seq()`

© 2012 Florian Schäfer

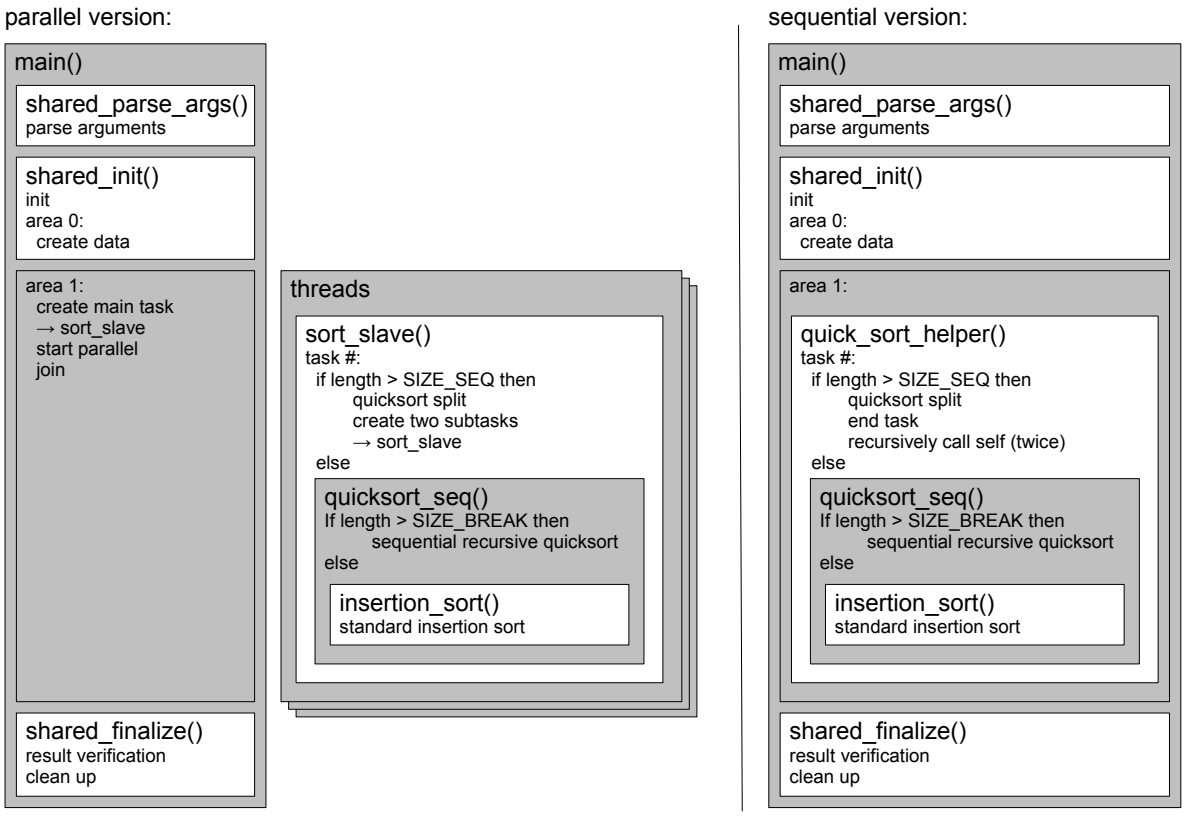


Figure 1: Quicksort example structure

- **shared_all.c** – the functions shared by all example (quicksort and merge-sort): `shared_init()`, `shared_finalize()`, `shared_parse_args()` and `insertion_sort()`

The quicksort example uses two areas, one sequential area (ID: 0) at the beginning used for creating the random data and one parallel area (ID: 1) which is used to execute all tasks in parallel. In the sequential version both areas are sequential but area IDs and task IDs are the same.

3 Methodology

The methodology used for verification is based on annotations added to the program that produce a runfile that contains data about the program run. The annotation statements are provided by the *HEAP API* library. The runfile is then analyzed by the analysis tool. The current version assumes that the annotations are added manually but any tool that

automatically parallelizes a program should have little difficulty to add these statements, since both annotations and parallelization are required in the same places. That means, the places where a parallelization tool needs to change the program's code to split the program into multiple parallel sections are the same places where the annotation statements need to be added in order to track the program's state before and after a parallel section. Consequently, the information required to parallelize a program is sufficient to also add the annotations.

Coverage analysis is performed by checking whether every checkpoint in the program has been encountered. If a checkpoint is encountered the surrounding code has been executed. By placing a checkpoint in every branch of the code, this allows to verify that all branches have been executed. In order to perform that check, the *HEAPAnalyze* tool requires the checkpoint statements in the program code, a structure file giving the analyzer a list of all checkpoints and possibly multiple resulting runfiles to check. The later might be required because depending on the program's structure it might not be possible to visit every branch of a program with a single run – especially if error conditions are to be checked. In this case, a successful standard run plus several runs to cover corner cases might be required to achieve full coverage. For this case multiple runfiles can be provided to the analysis tool and coverage will be calculated over all of them. That is, a checkpoint is marked as visited if it was visited by at least one of the runs.

The initial setup for verification and performance analysis uses the machine's native runtime environment and annotations to the program to produce the required data. Consequently, the design driver preparation consists of adding annotations to the program and modifying it in such a way that it can be run in the native environment. Actually, any runtime environment could be used as long as facilities for file I/O and precision timers are present.

The following steps need to be performed in order to prepare the design driver, assuming that the program exists in a parallelized version and a sequential version:

- **adapting program to the runtime environment** – the program must be able to run in an environment compatible with the *HEAP API*
- **including the HEAP-API** – the *HEAP-API* must be included into the program before it can be used
- **parallelizing the program** – the parallelization is done automatically or manually
- **segmenting the program** – dividing the program into logical areas (each purely parallel or sequential) by adding corresponding statements
- **dumping data** – adding data statements to the program to dump input and output data for later comparison

- **annotating the program** – adding statements for checkpoints, assertions and possibly executions states
- **compiling & running** – running the program will produce the required tracking data
- **annotating sequential version** – adding similar annotation statements to the sequential version to obtain reference data
- **analyzing the results** – running the *HEAPAnalyze* tool with the previously obtained runfiles as arguments

In addition to the *HEAP-API* library, a C-compiler and the POSIX Thread Library (include files, library stubs and dynamic library) are required. The setup was tested with gcc under Linux and mingw under Windows XP.

4 Adaption the Program to the Runtime Environment

The *HEAP API* library, that is used to track the program's course and write the data into a runfile, requires a runtime environment that provides file I/O, dynamic memory allocation and a precision timer. The timer may be omitted if no performance analysis has to be performed.

The library itself was designed for a native linux environment and also runs well in a Windows environment. It may also be possible to use it in different environments as long as the required facilities are provided.

The example used in this document – the *quicksort* example – has been designed for the native runtime environment from the start and thus requires no further adjustment. Other programs might require additional libraries /functions and adjustments to data types to run correctly.

5 Including the HEAP-API

The first step in preparing the design driver is to include the *HEAP API* into the program and prepare it for use. The following steps have to be performed:

- **Include headers** – including the *HEAP_API.h* header file into the source file that needs to use the *HEAP-API* library. The include search path of the compiler might

require adjustments to find the header files. If only the tracking statements and not the parallelism facilities are to be used, one can include the *HEAP_reporting.h* header only.

- **Initialize library** – before any other call to the *HEAP-API* library, initialization of the library is required. This is performed by calling `heap_init(...)` or `heap_report_init(...)` respectively.
- **Finalize library** – before the program exits but after any other call to the *HEAP API* library, calling `heap_finalize(...)` or `heap_report_finalize(...)` is required to terminate the other threads, flush out cached reports and close the runfile.
- **Making Memory Allocation thread-safe** – The *HEAP API* library uses dynamic memory allocation with `malloc()` and `free()`. When used in multiple threads, additional measures may be required to ensure thread-safety. See section 5.1.
- **Include HEAP-API library in compile** – the *HEAP-API* library files have to be included into the build. The library might need building itself – depending on the form of distribution. The build also has to include the pthread library (libpthread) if the parallelism facilities of the HEAP API are to be used. See section 11 for details.

The report only functions `heap_report_init` and `heap_report_finalize` can be used if only the reporting part of the *HEAP API* is used. If external parallelism facilities are used, the thread id of each thread must either be supplied manually by selecting the corresponding reporting functions or a function must be supplied to the library that returns the current thread id (assumed to be $0 - (n - 1)$ for n threads). The later one is done by calling `heap_set_get_thread_id_func(...)` (see section 5.2). Furthermore, thread-safe memory allocations may be an issue since the reporting library makes use of dynamic memory allocation. See section 5.1.

In the example program, library management is done in the shared functions in *shared_all.c*:

```
// shared_all.c

...
#include <HEAP_API.h>
...

...
void shared_init(){
  ...
```

```
  heap_init( g_para_runfile, g_para_num_threads );
  ...
}

void shared_finalize(){
  ...
  heap_finalize();
  ..
}

...
```

No additional means are necessary as the example uses the library default parallelism facilities.

5.1 Thread-safe Memory Allocation

The *HEAP API* library uses dynamic memory allocation with `malloc` and `free`. Depending on their implementation, these functions may not be thread-safe. If they are, no further actions is required (both the default Unix environment and the default windows environment has a thread-safe implementation). If they are not, the *HEAP API* provides a mutex based locking mechanism to ensure thread-safety:

All *HEAP API* functions surround calls to `malloc()` and `free()` with calls to `heap_mem_lock()` and `heap_mem_unlock()`:

```
  ...
  heap_mem_lock();
  new_cache = (heap_out_cache*)malloc(sizeof(heap_out_cache));
  buffer = (char*)malloc(sizeof(char)*length);
  heap_mem_unlock();
  ...
```

If thread-safety is not guaranteed, the calling program must do the same.

If the API's parallelism facilities are used, these two functions are used to lock and unlock a mutex and thus ensure thread-safety, assuming that the rest of the program does the same. As mentioned before, these functions can be ignored if malloc and free are thread-safe. If a different facility for parallelism is used and thread-safety cannot be

guaranteed, the `heap_mem_lock()` and `heap_mem_unlock()` functions must be set to custom functions that perform mutex locking. Use `heap_set_mem_lock_func(...)` and `heap_set_mem_unlock_func(...)` to do so. See section 15 for function details.

5.2 External Parallelism Facilities

If the *HEAP API's* internal parallelism facilities are used for parallelism, several additional configuration steps must be performed:

- **initializing external facilities** – initializing the external parallelism facilities. Refer to your facility's documentation for details.
- **providing a memory access mutex** – this step can be omitted if `malloc()` and `free()` are thread-safe. Otherwise create a mutex and two functions, one that locks the mutex and one that unlocks it. Provide these functions to the *HEAP API* by calling `heap_set_mem_lock_func()` and `heap_set_mem_unlock_func()`. See section 15 for details on functions.
- **providing an internal mutex** – this step is always required in multi-threaded environment. Create a mutex (different from the memory access mutex!) and two functions, one that locks the mutex and one that unlocks it. Provide these functions to the *HEAP API* by calling `heap_set_internal_lock_func()` and `heap_set_internal_unlock_func()`. See section 15 for details on functions.
- **providing a thread ID function** – various reporting statements also report the thread ID of the calling thread. This ID can either be provided manually by using the corresponding function version (`<function name>_t()`, see section 15 for details) or automatically by the *HEAP API*. For the later case to work the *HEAP API* requires a callback function that provides it with this ID. Use `heap_set_get_thread_id_func()` to set this function.
- **finalizing external facilities** – finalizing the external parallelism facilities after the program is done. Refer to your facility's documentation for details.

6 Parallelization

The parallelization step is not part of the verification. With regard to this aspect, please check the respective documentation for the parallelization efforts. This documentation only deals with manual parallelization using the *HEAP API's* facilities.

The *HEAP API* C-library provides a range of functions to simplify the parallelization – both manual and automatic. Currently, management for pool-based and consumer-producer parallelism is provided.

The provided facilities for parallelism work on a pool based task model. New tasks are added to the pool by calling `heap_add_task(...)`. Idle threads automatically fetch tasks from the pool and execute them. By only providing as many tasks as there are threads a consumer-producer style parallelism can be achieved. Additional data ports /pipes are provided for that scenario.

6.1 Pool-based Parallelism

In the case of pool-based parallelism, the work is broken down into several tasks. All information required to perform a task is bundled and stored in a list. Each thread fetches a task from the list and performs it. Once it is done, it fetches a new task from the list. This continues until the list is empty and all threads are idle.

The main thread can join the work force and wait for all tasks to be completed by calling `heap_join()`. The function returns once all tasks have been completed. Thread-safe memory allocation can be achieved by using `heap_mem_lock()` and `heap_mem_unlock()` to ensure compatibility with the *HEAP-API* library in case that `malloc()` and `free()` are not thread-safe by themselves. Details on the thread-safety issue can be found in section 5.1.

The steps required to set up a pool-based parallel program are:

- **reporting input data** – report the input data for this program section. See section 8.
- **reporting beginning of parallel area** – report the beginning of a parallel area. See section 7
- **setting up initial task(s)** – create one or more tasks that are to be performed. Note that you can add additional tasks during the execution of tasks (e.g. for recursive algorithms)
- **writing a task function** – the task function is the function that is called by a thread to perform a task. It is not necessary for one function to be able to perform all tasks, you can have as many task functions as required. The function used is part of the task definition.
- **reporting beginning and end of tasks** – each task function should report the beginning and end of a task using the corresponding annotation statements. See section 7.1.

- **calling join()** – after setting up the initial task(s), the main thread should join the remaining threads and contribute to the work. `heap_join()` does that and returns once all tasks are completed
- **reporting end of parallel area** – report the end of the previously started parallel area. See section 7
- **reporting output data** – report the output data for this program section. See section 8.

Functional details are described in section 15.

A template for a parallel area using the pool-based facility looks like this:

```
// area_id = "area_1"
heap_report_data( "area_1", "input", (void*)input,
                  sizeof(int)*input_size );
heap_report_start_parallel( "area_1" );
// set up tasks
for( i=0; i<10; i++ ){
  heap_add_task( task_func, (void*)i );
}
// join work force and wait for completion
heap_join();
// end of area
heap_report_end_parallel( "area_1" );
heap_report_data( "area_1", "output", (void*)output,
                  sizeof(int)*output_size );

...

void task_func(int id, void *void_arg){
  int task_number = (int)void_arg;
  // task id
  char task_id[32];
  sprintf(task_id, "task_%i", task_number );

  heap_report_start_task( task_id );
  // do work
  ...
  heap_report_end_task( task_id );
}
```

This template does all the previously mentioned steps: reporting the input

and output data with `heap_report_data()`, reporting the parallel area with `heap_report_start_parallel()` and `heap_report_end_parallel()`, setting up the initial task(s) with `heap_add_task`, defining the task function `task_func()` and calling `heap_join()`.

The quicksort example introduced in section 2 uses a slightly more complex way to communicate task information, a task description structure:

```
// quick_sort.c
...
typedef struct{
int *start;
int length;
int task_id;
}sort_task;
...
sort_task *make_sort_task( int *start, int length, int task_number ){
  heap_mem_lock();
  sort_task *result = (sort_task*)malloc(sizeof(sort_task));
  heap_mem_unlock();
  result->start = start;
  result->length = length;
  result->task_number = task_number;
  return result;
}
...
int main(int argc, char **argv){
  sort_task *task=NULL;
  ...
  heap_report_start_parallel( "area_1" );
  task = make_sort_task( data, g_para_data_size, 1 );
  start = heap_time();
  heap_add_task( sort_slave, (void*)task );
  heap_join();
  end = heap_time();
  heap_report_end_parallel( "area_1" );
  ...
}
...
void sort_slave(int id, void *parm){
  int *left, *right, *pivot_pos;
  int pivot, temp;
  sort_task *sub1, *sub2, *task = (sort_task*)parm;
```

```
  int *start = task->start;
  int length = task->length;
  int task_number = task->task_number;
  char task_id[32];
  sprintf(task_id, "task_%i", task_number );
  heap_report_start_task( task_id );
  ...
}
```

A `sort_task` structure is created and filled with information. This structure – or rather its address – is used as argument for `heap_add_task()` and is passed to the task function `sort_slave()`.

6.2 Consumer-Producer style Parallelism

The JPEG example that is part of the *HEAP API* distribution shows a consumer-producer style parallelism. This is achieved by creating one task per thread and forwarding data from one task (producer) to another task(consumer). Data forwarding can be done by using data ports. Data ports allow one thread /task to insert data into the port which can store up to a certain number of data items (defined at creation time). Another thread /task can then retrieve the stored data.

- **creating a data port** – create a new data port by calling `heap_port_create()`. Each port has an ID, a given size for data items, an a size (in number of data items)
- **finding an existing port** – since ports must be shared between threads to be of any use, one can retrieve an existing port by its ID using `heap_port_find()`
- **attaching a report data stream** – unlike in the previous pool-based example, the data in this case does not exist as a large block at any one time. Therefore, the block data reporting functions used so far cannot be used. Instead the stream data reporting functions must be used. See section 8.2 for details. The data sent through a data port can be automatically reported by attaching a report data stream to it by calling `heap_port_attach_stream()`.
- **writing data to a port** – write data to a port by calling `heap_port_write()` or `heap_port_write_multiple()`
- **reading data from a port** – read previously written data by calling `heap_port_read()` or `heap_port_read_multiple()`
- **closing a data port** – data ports are automatically closed when `heap_finalize()` is called. But they can also be closed manually by calling `heap_port_close()`.

Whether an attached report data stream is also closed can be defined when the stream is attached by setting the `close_stream` flag.

The JPEG example main:

```
...
void thread_2( int id, void *arg ){
  In<TBlocks> ig_1("p12");
  Out<TBlocks> og_1("p23");
  ND_2 nd2( 2, new ND_2_Ports( ig_1, og_1 ) );
  heap_report_start_task("task_2");
  nd2.main();
  heap_report_end_task("task_2");
}
...
int main(int argc, char **argv){
  // set up ports
  heap_port_handle port_0_5 = heap_port_create( "p05",
    sizeof(struct THeaderInfo), g_port_size );
  heap_port_handle port_1_2 = heap_port_create( "p12",
    sizeof(struct TBlocks), g_port_size );
  heap_port_handle port_2_3 = heap_port_create( "p23",
    sizeof(struct TBlocks), g_port_size );
  heap_port_handle port_3_4 = heap_port_create( "p34",
    sizeof(struct TBlocks), g_port_size );
  heap_port_handle port_4_5 = heap_port_create( "p45",
    sizeof(struct TPackets), g_port_size );
  // set up data streams
  heap_stream_handle stream_0_5 =
    heap_report_data_stream( "area_1", "stream_port_0_5", -1 );
  heap_stream_handle stream_1_2 =
    heap_report_data_stream( "area_1", "stream_port_1_2", -1 );
  heap_stream_handle stream_2_3 =
    heap_report_data_stream( "area_1", "stream_port_2_3", -1 );
  heap_stream_handle stream_3_4 =
    heap_report_data_stream( "area_1", "stream_port_3_4", -1 );
  heap_stream_handle stream_4_5 =
    heap_report_data_stream( "area_1", "stream_port_4_5", -1 );
  // attach streams to ports
  heap_port_attach_stream( port_0_5, stream_0_5, 1 );
  heap_port_attach_stream( port_1_2, stream_1_2, 1 );
  heap_port_attach_stream( port_2_3, stream_2_3, 1 );
```

```
  heap_port_attach_stream( port_3_4, stream_3_4, 1 );
  heap_port_attach_stream( port_4_5, stream_4_5, 1 );
  // start
  double start_time = heap_time();
  heap_report_start_parallel("area_1");
  heap_add_task( thread_0, 0 );
  heap_add_task( thread_1, 0 );
  heap_add_task( thread_2, 0 );
  heap_add_task( thread_3, 0 );
  heap_add_task( thread_4, 0 );
  heap_add_task( thread_5, 0 );
  heap_join();
  heap_report_end_parallel("area_1");
}
```

In the first step one task function is defined per thread (only one shown). Then the data ports are created, corresponding report data streams are created and attached and the tasks are added to the task pool. `heap_join()` finally completes the consumer-producer setup. Note that neither the data ports nor the data streams are closed explicitly. This example relies on the automatic closing feature when the library is finalized.

Note that it is not necessary for a consumer-producer setup to form a simple linear chain with respect to data flow. More complicated setups are also possible. The *HEAP API* facilities also allow for multiple threads to perform the same station in the chain. Although, one needs to keep in mind that the result order might change in that case.

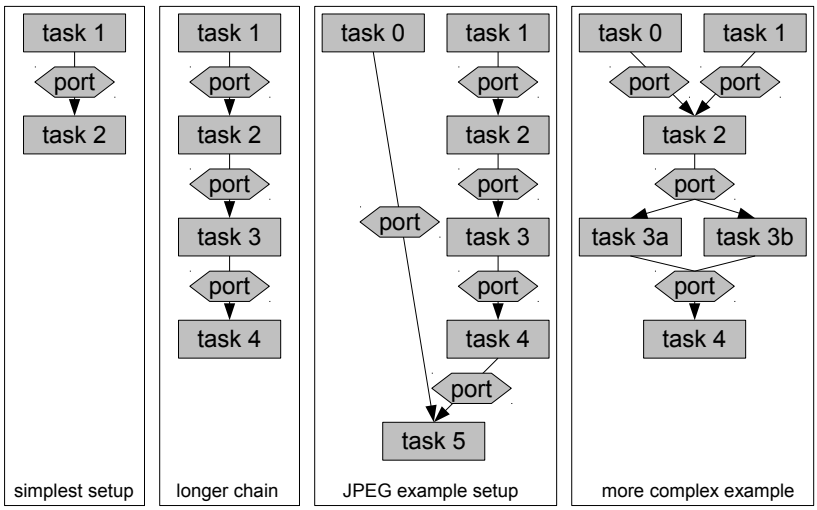


Figure 2: Consumer-Producer Data Chains

7 Program Segmentation

The first step of annotation is to segment the program. That is to divide the program into several parts – called areas. Each area is either sequential or parallel. All code between areas – that is between for example `heap_report_end_sequential` and `heap_report_start_parallel` – is ignored for the purpose of analysis and should be used only for dumping data. Each pass though an area is analyzed separately.

Whenever a program switches between sequential and parallel execution, a new area should be introduced. Dividing any sequential or parallel segments into multiple areas is not necessary but may be useful to track intermediate data or for a more fine grained performance analysis. An exception is a sequential reference program. In order to assess the correctness of the parallel program by comparing it to its sequential version, both versions must of course use the same segmentation, checkpoints and data dumps. See section 12 for details. Nested areas are not supported.

In addition to areas, tasks can be declared. Tasks are code segments inside a parallel area and allow to keep track of what each thread is doing. They are used to uniquely identify code segments and passes through code segments in a parallel area to allow comparison to a sequential version. See section 7.1.

The segmentation itself, is performed by adding corresponding statements to the program that report the beginning and end of each area. The statements are:

- **heap_report_start_sequential(...)** – to start a sequential area
- **heap_report_end_sequential(...)** – to end a sequential area
- **heap_report_start_parallel(...)** – to start a parallel area
- **heap_report_end_parallel(...)** – to end a parallel area
- **heap_report_start_task(...)** – to start a task in an area
- **heap_report_end_task(...)** – to end a task in an area

The API currently makes no effort to verify the correctness to calls to these function. One could have several areas with the same ID or several consecutive calls to start an area or calls to end an area with an ID different from the last call to start an area. Such a program would compile and run, but the analysis tool will complain when reading these files.

An example segmentation is given in figure 3.

The example program contains two areas: A sequential area to create the input data and a parallel area comprising the actual sorting:

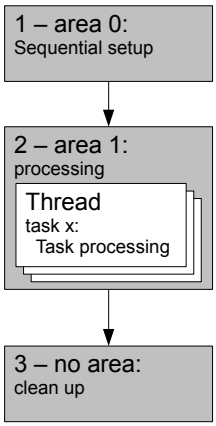


Figure 3: Quicksort example segmentation

```
// shared_all.c
...
void shared_init(){
  ...
  heap_report_start_sequential("area_0");
  ...
  srand( g_para_seed == -1 ? time(NULL) : g_para_seed );
  for(i=0;i<g_para_data_size;i++)
    data[i] = rand();
  heap_report_end_sequential("area_0");
  heap_report_data( "area_1", "input_1", (void*)data,
                    sizeof(int)*g_para_data_size );
}
...
```

The sequential area is surrounded by `heap_report_start_sequential()` and `heap_report_end_sequential()`.

```
// quick_sort.c

...

int main(int argc, char **argv){
  ...
  heap_report_start_parallel( "area_1" );
```

```
  task = make_sort_task( data, g_para_data_size, 1 );
  start = heap_time();
  heap_add_task( sort_slave, (void*)task );
  heap_join();
  end = heap_time();
  heap_report_end_parallel( "area_1" );
  ...
}

...
```

The parallel area is surrounded by `heap_report_start_parallel()` and `heap_report_end_parallel()` and actually comprises several thread instances running `sort_slave()`. Each instance performs tasks as explained below.

The code segment after the parallel area is not marked as belonging to any area and will be ignored performance wise which is desired since it only contains debugging checks.

7.1 Tasks

The work performed by the threads in area 1 is divided into tasks. In the example, each task consists of splitting a certain part of the data set into two parts according to the quicksort algorithm or completely sorting the part – depending on the size of the part (split if `size > SIZE_SEQ`). These tasks are performed in parallel and not in a predefined order. This poses some problems when trying to compare the course of the program between different runs or even between a parallel and a sequential run. The solution to this problem, is the declaration of tasks with the *HEAP API*. This assigns each task a unique ID which must be the same over all runs that are to be compared.

In the example, the first task is given the number 1 and a HEAP API task id of "task_1". Whenever a task (with task number x) splits the work and spawns two new tasks, the left task gets the number $2x$ and the right one gets $2x + 1$. The resulting task numbers look like this:

These task numbers are always the same, no matter in which order the tasks are executed. This allows the analysis to compare them later to other runs.

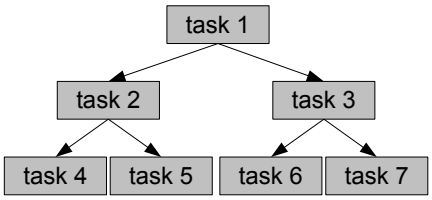


Figure 4: Quicksort example task numbering

8 Dumping Data

Data dumps are required by the analysis tool to verify computational correctness. This is achieved by comparing the data of test run against the data of a (sequential) reference run. Data comparison is done on a per-area basis. Each data set has a data ID that allows it to be uniquely identified if several data sets are dumped per area. That means there can be at most one set of data per data ID, area and pass (each pass through an area can of course have its own set of data). It is not necessary to dump the same data twice as both output of an area and input of the next area.

There are two ways to dump data: either as a continuous block or as a stream. Either way, the resulting data will be dumped in a additional file next to the runfile. This file named similar to the runfile: `<runfile_name>.data<continous_number>.<runfile_ext>`.

8.1 Block Data

For data to be dumped as a whole block, the entire data set must reside continuously in memory. Furthermore, the process of dumping the data involves file I/O and may be time-consuming. It is therefore highly recommended to dump all data outside of any area to reduce the impact on measured performance.

Dumping data is performed by invoking;

- **heap_report_data(...)**

The data dumping statements can be disabled and re-enabled independently of other reporting statements using the `heap_report_data_enable` function of the *HEAP-API*. Alternatively, they can be controlled together with the other reporting statements using `heap_report_enable`.

Data dumping in the quicksort example:

```
// shared_all.c
...
void shared_init(){
  ...
  heap_report_end_sequential("area_0");
  heap_report_data( "area_1", "input_1", (void*)data,
                    sizeof(int)*g_para_data_size );
}

void shared_finalize(){
  ...
  heap_report_data( "area_1", "output_1", (void*)data,
                    sizeof(int)*g_para_data_size );
  ...
}
...
```

Note that the data is dumped as input and respectively output of "area 1" – the parallel area. The input data could also be dumped as output of "area 0" but since the critical area to analyze is the parallel area, this setup makes more sense. Also note that the data dumps are placed outside of area to reduce the impact on performance analysis.

8.2 Stream Data

Alternatively, data can be dumped continuously throughout the course of the program. This is ideal for tracking streaming data. To reduce the impact on performance measurements, the tracked stream data is tracked in memory and if possible only dumped outside of areas.

Unlike the block data tracking, several steps are required to track streaming data:

- **create report data stream** – create a new report data stream with `heap_report_data_stream()`.
- **write data to report data stream** – this can be done manually by calling `heap_report_add_to_stream()` or automatically by attaching the report data stream to a data port with `heap_port_attach_stream()`. In the later case, all data written into the data port is automatically tracked.

- **close the report data stream** – This can be done manually by calling `heap_report_close_stream()` or automatically when closing an attached data port or automatically when the *HEAP API* is finalized.

Data Stream Dumping in the JPEG example:

```
// mjpeg_parallel.cpp:
...
int main(int argc, char **argv){
  ...
  heap_port_handle port_0_5 = heap_port_create( "p05",
    sizeof(struct THeaderInfo), g_port_size );
  ...
  heap_stream_handle stream_0_5 = heap_report_data_stream( "area_1",
    "stream_port_0_5", -1 );
  ...
  heap_port_attach_stream( port_0_5, stream_0_5, 1 );
  ...
}

// port.h
...
template<typename DATATYPE>
  class Out{
    public:
      heap_port_handle handle;
    public:
      explicit Out( const char *port_id )
        :handle(heap_port_find(port_id)){
      }
      ...
      void write( const DATATYPE &data ){
        heap_port_write( handle, &data );
      }
  };
...
```

The main program first creates the data ports, the corresponding report data streams and attached them to one another. Each task then uses the template classes `In<T>`(not shown) and `Out<T>` to access the data ports via `heap_port_find()` and read /writes data to them using `heap_port_read()`(not shown) and `heap_port_write()`. The written data

is automatically forwarded to the attached report data stream. Closing of the data ports and report data streams is done automatically when the library is finalized.

9 Program Annotation

Program annotations help the analysis tool in verifying correctness, determining program flow and calculating program performance. The following classes of annotations exist:

- **state** – Tells the analyzer whether a thread is idle, working or synchronizing(waiting). If the *HEAP-API* facilities for parallelism are used, the thread state is set automatically. Otherwise, it has to be set manually. The relevant function is `heap_report_state(...)`.
- **assertion** – Assertions are checks performed by the program at runtime and their outcome reported. This is one way to perform data integrity checks. The relevant functions and macros are `heap_report_assert(...)`, `HEAP_ASSERT_MSG(...)` and `HEAP_ASSERT(...)`.
- **checkpoints** – Checkpoints can be used to verify that the program flow of two versions (parallel version and sequential reference) are identical and/or to perform coverage checks by ensuring that all checkpoints in a program are encountered (possibly over several runs). Unordered checkpoints only verify coverage, ordered checkpoint verify coverage and code execution order. Functions used are `heap_report_chkpnt_unordered(...)` and `heap_report_chkpnt_ordered(...)`.

All functions have an alternative version where the thread id can be explicitly given. See section 15

9.1 State

The thread state is required by the performance analysis to calculate idle times and synchronization overhead. Whenever a thread switches from one state (idle, working, synchronizing, terminated) to another, a corresponding statement is written to the runfile. If the *HEAP API's* parallelism facilities are used, this state is set automatically and no further work is required. If another parallelism facility is used, the thread state must be updated manually using `heap_report_state()`.

Relevant function:

- **updating the thread state** – `heap_report_state()`

All of the example used in the *HEAP API* use the internal parallelism facilities and therefore never set the thread state manually. But the pseudocode for a thread's idle function could look like this:

```
void thread_idle( int thread_id ){
  while(TRUE){
    lock_mutex( task_list_mutex );
    while( task_list_empty() ){
      heap_report_state_t( l_id, HEAP_STATE_IDLE );
      wait_for_signal( signal_work_ready, task_list_mutex );
    }
    task = get_task();
    unlock_mutex( task_list_mutex );
    if( task == kill_thread ){
      heap_report_state_t( l_id, HEAP_STATE_TERMINATED );
      return;
    }else{
      heap_report_state_t( l_id, HEAP_STATE_WORK );
      task( thread_id );
    }
  }
}
```

This pseudocode checks whether there are tasks available. If not, the thread's state is set to *IDLE* and the thread wait for new tasks to arrive. Once a task is available the state is set to *WORK* and the tasks executed. This continues until the thread is signaled to terminate itself.

9.2 Assertions

The simplest form of verification annotation is an assertion. An assertion comprises a condition that is assumed to be always true when the assertion statement is reached plus an optional message that is displayed should the assertion not hold true. Unlike most assertion facilities the assertion of the *HEAP API* do neither abort the execution of a program nor raise an exception or display a message at runtime. Instead the results of assertions – both negative and positive – are written to the runfile for later analysis.

Assertions are also the only verification annotations that do not require a reference to compare to.

The macros used to check an assertions are:

- `HEAP_ASSERT_MSG( assertion, message )` – the verbose version: requires the programmer to provide a message. *assertion* serves as the condition checked and as display string. During analysis both the assertion and the message are displayed.
- `HEAP_ASSERT( assertion )` – simple version, no message is given.

Alternatively, the reporting function can be used directly:

- `void heap_report_assert( int success, const char *assertion, const char *message )` – set *success* to zero if the condition failed and non-zero otherwise. The *assertion* string is required and should be equal to the condition checked. The *message* is optional.

All functions have an alternative version where the thread id can be explicitly given. See section 15

Example:

```
// shared_all.c

...
void shared_finalize(){
  ...
  error_found = -1;
  for( i=1;i<g_para_data_size;i++){
    if( data[i] < last ){
      printf( "ERROR: [%i]=%i > [%i]=%i\n", i-1, last, i, data[i] );
      error_found = i;
     }
    last = data[i];
  }
  HEAP_ASSERT_MSG( error_found == -1, "final sort check" );
  ...
}
```

9.3 Checkpoints

Checkpoints are annotations that simply state that the program passed a specific point in the code at a certain time and therefore in a certain order. They serve two purposes: First,

coverage – if a run of the program never passed a certain point, the code around that point was not executed and consequently not verified. Second, order – if the parallelization of code was successful and error-free, one can expect that code inside an area is executed identically both in the sequential version and the parallel version. Checkpoints can be used to verify that.

Both kinds of checkpoints – ordered and unordered – are used for coverage analysis but only ordered checkpoints are considered when verifying the course the execution took through the code.

In order for checkpoints to work, each checkpoint needs to have its own ID assigned. The analysis tool cannot tell apart two checkpoints using the same ID. It can tell apart an ordered and an unordered checkpoint with the same ID though. Therefore, overlap between ordered and unordered checkpoints is not an issue. There might be special corner cases, in which it might be reasonable to invoke the same checkpoint from two different point by using the same ID. This is supported, though no such corner case has been encountered, yet.

9.3.1 Unordered Checkpoints

The simplest type of checkpoint is an *unordered checkpoint*. One could also call it simple checkpoint. It is only used to check whether this segment of code has been executed. This type of checkpoint does not need to be present in the reference runfile and consequently does not need to be introduced into the sequential version of the program. It is allowed there, though it will not have any effect.

Each checkpoint has an ID. IDs can be chosen freely but should be unique. Sharing IDs between checkpoints is allowed but may compromise coverage analysis as the analyzer only verifies that any checkpoint with a given ID is encountered, not that all of them are encountered (since the analyzer cannot tell them apart).

Library function:

- `void heap_report_chkpnt_unordered( heap_chkpnt_id chkpnt_id )` – reports an unordered checkpoint in the current thread with ID *chkpnt_id*.

All functions have an alternative version where the thread id can be explicitly given. See section 15

Example:

```
// shared_all.c

...
void shared_init(){
  ...
  heap_report_start_sequential("area_0");
  data = (int*)malloc(sizeof(int)*g_para_data_size);
  srand( g_para_seed == -1 ? time(NULL) : g_para_seed );
  heap_report_chkpnt_unordered("u1");
  ...
}
...
```

This example shows an unordered checkpoint in a sequential area. Using an ordered checkpoint in this place would provide no advantage since this code is only executed once and there is no branching.

9.3.2 Ordered Checkpoints

An advanced version of the unordered checkpoint is the *ordered checkpoint*. It serves the same purpose as the unordered checkpoint but is also used for checking correctness. When verifying correctness, the order of ordered checkpoints encountered inside an area or task is verified to be the same as in the reference runfile. That is, they can be placed in a sequential area without any task definition and should perform fine as without parallelism their order should be the same during every execution. In a parallel area however, that is not the case: due to the parallelism their order within the area is partially random. Only their order within a single thread and for a certain task is defined. A task in this context referring to a certain part of code executed on a certain data set: different tasks may execute different code on different data sets or the same code on different data sets or different code on the same data set. Additionally, the order of tasks executed might also be partially random. In this case, the analysis tool requires the tasks to be explicitly reported as described in section 7.1. It is vital that the same tasks in the parallel version and the sequential reference version of the program use the same task IDs, otherwise comparison will fail.

By placing ordered checkpoints in every branch of the code, one can guarantee that two versions of the program (parallel and sequential) take the same route through the code: Otherwise, the checkpoints encountered and their order would differ.

Note that the ordered checkpoints use an ID system separate from the unordered checkpoints. That means having ordered and unordered checkpoints with the same IDs is not

a problem.

Library functions:

- `void heap_report_chkpnt_ordered_t( int thread_id, int chkpnt_id )` – reports an ordered checkpoint in the current thread with ID *chkpnt_id*.

All functions have an alternative version where the thread id can be explicitly given. See section 15

Example:

```
// quick_sort.c
...
void sort_slave(int id, void *parm){
  ...
  heap_report_start_task( task_id );
  ...
  if( length <= SIZE_SEQ ){
    heap_report_chkpnt_ordered("o1");
    quicksort_seq( start, length );
    heap_report_end_task( task_id );
    return;
  }
  heap_report_chkpnt_ordered("o2");
  ...
  heap_report_end_task( task_id );
  ...
}
...
```

This example shows a piece of branching code inside a task. In both branches an ordered checkpoint with a different ID is placed. This allows the analyzer not only to verify that both branches have been executed but also to verify that the same branches are taken in each task compared to the corresponding tasks in the reference run.

10 Structure File

The last piece required for coverage analysis is the structure file. The structure file tells the analyzer which areas a program has and gives a list of all ordered and unordered

checkpoints.

Each information is given in a single line:

- **areas** – this line tells the analyzer which areas there are. This is a comma-separated list of the area- IDs
- **checkpoints** – this line gives a comma-separated list of the IDs of unordered checkpoints.
- **ordered checkpoints** – same as above but for ordered checkpoints

Note that these IDs must not start or end with whitespace-characters (space, tab) as they will be stripped away.

Example:

```
// quick_sort_structure.txt
areas: area_0, area_1
checkpoints: u1
ordered checkpoints: o1, o2, o3, o4, o5, o6, o7
```

An example for a program with two areas ("area_0" and "area_1"), 1 unordered checkpoint (ID: "u1") and 7 ordered checkpoints (IDs "o1" through "o7" (inclusive)).

11 Compilation & Running

Once the program has been segmented and annotated, it needs to be compiled and run to produce the runfile which is the input for the analysis tool. The *HEAP-API* library was designed for use in C-programs compiled with gcc. It also requires the POSIX Thread library (libpthread) which should come with every linux installation but must be obtained separately if a Windows installation is to be used as host OS (recent version of mingw comes with the pthread library). A windows port of the pthread-library can be found online under the name *pthreads-win32*. The pthread library may be omitted if only the reporting part of the *HEAP API* library is used.

Makefiles for the HEAP-API are provided which have been tested in linux environment with gcc and in a Windows environment with mingw. Other environment might work but may require adjustments. The makefile provided with the example programs may be used as a template for the program compilation.

Running the compiled program will produce a runfile (unless reporting has been disabled) named *runfile.txt* by default. Additionally, data files will be produced that have a similar name as the runfile: *¡runfile_name¿.data¡continous_number¿.¡runfile_ext¿.*

12 Annotation of the Sequential Version

For verifying the correctness of a parallel program, a reference runfile is required. This reference runfile is obtained by annotating, compiling and running the original sequential version of the program. In order for the two runfiles to be compared, both versions of the program must be annotated analogically. That means, the same segmentation must be used including the same task IDs. The sequential version obviously only contains sequential areas. These have to have the same IDs as the parallel areas, though. If the order of checkpoints encountered is to be verified, both versions also have to have the same ordered checkpoint statements at the same places. Assertions are generally not compared and can be omitted in the sequential version.

It is possible to have the analysis tool compare two parallel versions with each other. Whether this allows any meaningful conclusions concerning the correctness of either version must be decided on a per case basis.

The sequential variant of the example program is *quick_sort_seq*:

```
// quick_sort_seq.c
...
int main(int argc, char **argv){
  ...
  heap_report_start_sequential("area_1");
  start = heap_time();
  // do the sorting, same task IDs as in parallel version
  quick_sort_helper( 1, data, g_para_data_size );
  end = heap_time();
  heap_report_end_sequential("area_1");
  ...
}
```

The main difference is that “area 1” is annotated as a sequential area not a parallel one as in the parallel version. Furthermore, the actual sorting is done with a sequential function not parallely executed tasks.

© 2012 Florian Schäfer

13 Adaption to other Environments

The chosen environment – namely the machines native runtime environment – is the easiest to use and very flexible program wise but it is not the best with regard to performance analysis as the embedded annotation alter the program's execution time. Furthermore, the native machine may behave differently than the intended environment which may cause wrong results, e.g.: A parallel program running fine on a x86 linux environment may fail on an ARM-based SOC.

The ideal environment would be a simulator with precise representation of hardware timing and behavior. Furthermore, such an environment may not provide the necessary facilities for the *HEAP API* library. It would be best to incorporate the tracking and data dumping into the simulator and only annotate the program by a description file for the simulator which describes what to track when. Unfortunately, such a simulator would have to be constructed (or at least adapted) for each environment to be used.

14 HEAPAnalyze

The analysis tool of the HEAP test suite is called *HEAPAnalyze*. It is a Java based command line tool. The tool takes a list of commands, one or more runfiles and if necessary a structure file and a reference runfile. All given runfile will be parsed and assuming that succeeded, the commands will be executed – not necessarily in the order given.

14.1 Options

The *HEAPAnalyze* tool has the following command line syntax:

```
HEAPAnalyze <command> [<command> ...] <run file to check>
            [<run files to check> ...] [-reference <reference file>]
            [-structure <structure file>]
```

The following commands are available:

- **ASSERT** – checks whether all assertions succeeded. See section 14.2 for details.

- **DATA** – performs data comparison, requires reference runfile. See section 14.3 for details.
- **ORDER** – performs order comparison, requires reference runfile. See section 14.4 for details.
- **VERIFY** – all three verification steps mentioned above together, requires reference runfile. Same as DATA, ORDER and ASSERT combined.
- **COVERAGE** – performs coverage analysis, requires structure file. See section 14.5 for details.
- **PERFORMANCE** – performs performance analysis, benefits from a reference runfile. See section 14.6 for details.
- **ALL** – performs all checks and analyses, requires both a reference runfile and a structure file
- **HELP** – shows help
- **VERBOSE** – sets verbose level (0-4), default is 1

14.2 Assertion Checking

Assertion checking is performed by invoking the *ASSERT* command of the HEAPAnalyze tool. This command does not require any other files. The program runs through all reported assertions and checks that they have been reported as successful. An example run for the quicksort example at maximum verbose level looks like this:

```
> java -jar HEAPAnalyze.jar ASSERT VERBOSE 4 data/quick_sort_run.txt
Analyzing runfile to check ('data/quick_sort_run.txt')
  verifying
Verification of data/quick_sort_run.txt
  Checking assertions
  checking assertions in 'data/quick_sort_run.txt'
    checking ASSERTION: thread 0 @ 0.0538460000007035: error_found == -1
 - final sort check
      ok
    all assertions encountered are ok
  all checks successful

all file checks completed successfully
```

At lower verbose levels, less detailed output is generated. But errors will always be reported.

14.3 Data Verification

Data comparison is performed with the *DATA* command. The tool then goes through both the runfile to check and the reference runfile and compares the data entries for each pass through each area. A comparison run at maximum verbose level looks like this:

```
> java -jar HEAPAnalyze.jar DATA VERBOSE 4 data/quick_sort_run.txt
>   -r data/quick_sort_seq_run.txt
Analyzing runfile to check ('data/quick_sort_run.txt')
  verifying
Analyzing reference runfile ('data/quick_sort_seq_run.txt')
  verifying
Verification of data/quick_sort_run.txt
  Comparing data
    comparing 'data/quick_sort_run.txt' and 'data/quick_sort_seq_run.txt'
    processing area area_0 pass 1
    processing area area_1 pass 1
      comparing 'input_1' data
        match
      comparing 'output_1' data
        match
    all data matches
  all checks successful

all file checks completed successfully
```

Note, that it is an error if the runfile and reference file differ in the amount of areas or number of passes though each area. It is also an error if the runfile does not provide a data set that the reference provides. If only a reference set is missing, this will trigger a warning. If both are missing, it is assumed to be intentional.

14.4 Order Verification

Order verification is performed with the *ORDER* command. The tool then goes through both the runfile to check and the reference runfile and compares the entries for ordered checkpoint for each pass through each area and each task. A comparison run at maximum verbose level looks like this:

```
> java -jar HEAPAnalyze.jar ORDER VERBOSE 4 data/quick_sort_run.txt
```

```
>   -r data/quick_sort_seq_run.txt
Analyzing runfile to check ('data/quick_sort_run.txt')
  verifying
Analyzing reference runfile ('data/quick_sort_seq_run.txt')
  verifying
Verification of data/quick_sort_run.txt
  Checking order
    comparing 'data/quick_sort_run.txt' and 'data/quick_sort_seq_run.txt'
    processing area area_0 pass 1
      compared 0 ordered checkpoints in 0 tasks
    processing area area_1 pass 1
      compared 485 ordered checkpoints in 243 tasks
    all checkpoints match
  all checks successful

all file checks completed successfully
```

14.5 Coverage Analysis

Coverage analysis is performed by running the HEAPAnalyze tool with the *COVERAGE* command. The analysis tool then runs through all provided runfiles to check and verifies that all areas and checkpoints described in the structure file are encountered. A coverage check at maximum verbose level looks like this for the quicksort example:

```
> java -jar HEAPAnalyze.jar COVERAGE VERBOSE 4 data/quick_sort_run.txt
>   -s data/quick_sort_structure.txt
Analyzing runfile to check ('data/quick_sort_run.txt')
  verifying
Loading structure file ('data/quick_sort_structure.txt')

Coverage analysis
  collecting checkpoint data
    from data/quick_sort_run.txt
  performing integrity checks
  comparing to structure file

  area coverage:
    coverage: 2 / 2 =  100,00%

  unordered checkpoint coverage:
```

```
    coverage: 1 / 1 =  100,00%

  ordered checkpoint coverage:
    ordered checkpoint not encountered: o7
    coverage: 6 / 7 =  85,71%

  total checkpoint coverage:
    coverage: 7 / 8 =  87,50%
```

In this example, the checkpoint coverage is only $87,5\%$. That is because one checkpoint (*o7*) is located in a branch that covers a rather rare corner case and is not covered by this test.

14.6 Performance Analysis

Performance analysis is performed by running the HEAPAnalyze tool with the *PERFORMANCE* command. The analysis tool then creates a performance report for all given runfiles including the reference runfile (if provided) and finally a comparison between each runfile and the reference runfile.

Beware that most of the information compiled by the performance report is only meaningful if each thread has a compute core to itself. Otherwise the reported times do not correspond to actual CPU-time.

Due to the large amount of data produced, only a per-area report is generated by default. Additional information can be requested by adding suffixes to the command. Available suffixes are:

- **+TA** – adds per task reports
- **+TH** – adds per thread reports
- **+PA** – adds per pass reports

These suffixes can be combined: *PERFORMANCE+PA+TA* for example creates additional reports per task, per pass and per task and pass (and therefore potentially a huge report!).

For the quicksort example a simple performance analysis looks like this:

```
> java -jar HEAPAnalyze.jar PERFORMANCE VERBOSE 4 data/quick_sort_run.txt
```

```
>   -r data/quick_sort_seq_run.txt
Analyzing runfile to check ('data/quick_sort_run.txt')
  verifying
Analyzing reference runfile ('data/quick_sort_seq_run.txt')
  verifying

Performance analysis of reference data/quick_sort_seq_run.txt
  compiling performance report for data/quick_sort_seq_run.txt

================================== Reference ============================

Performance report:
runfile data/quick_sort_seq_run.txt: start @    0,00 ms
  WORK      :   59,67 ms = 100,00%
  total     :   59,67 ms

  in areas only: start @    0,00 ms
    WORK      :   29,66 ms = 100,00%
    total     :   29,66 ms

  area area_0: start @    0,40 ms
    WORK      :    3,42 ms = 100,00%
    total     :    3,42 ms

  area area_1: start @   19,08 ms
    WORK      :   26,24 ms = 100,00%
    total     :   26,24 ms



Performance analysis of data/quick_sort_run.txt
  compiling performance report for data/quick_sort_run.txt

================================== data/quick_sort_run.txt ==============

Performance report:
runfile data/quick_sort_run.txt: start @    0,00 ms
    IDLE      :   35,37 ms =  32,80%
    WORK      :   72,45 ms =  67,20%
    total     :  107,82 ms

    in areas only: start @    0,00 ms
      IDLE      :    4,29 ms =   9,38%
```

```
      WORK      :   41,41 ms =  90,62%
      total     :   45,70 ms

    area area_0: start @    0,53 ms
      IDLE      :    3,56 ms =  50,00%
      WORK      :    3,56 ms =  50,00%
      total     :    7,11 ms

    area area_1: start @   19,80 ms
      IDLE      :    0,73 ms =   1,89%
      WORK      :   37,85 ms =  98,11%
      total     :   38,58 ms



---------------------------------- Comparison --------------------------

Comparison report:
  overall:
    Speedup:     59,67 ms /   53,95 ms =    1,11
    Real-Time:   53,95 ms /   59,67 ms =   -9,6%
    CPU-Time:   107,82 ms /   59,67 ms =  +80,7%

  in areas only:
    Speedup:     29,66 ms /   22,85 ms =    1,30
    Real-Time:   22,85 ms /   29,66 ms =  -23,0%
    CPU-Time:    45,70 ms /   29,66 ms =  +54,1%

  per Area:
    area_0 overall:
      Speedup:      3,42 ms /    3,56 ms =    0,96
      Real-Time:    3,56 ms /    3,42 ms =   +4,0%
      CPU-Time:     7,11 ms /    3,42 ms = +108,0%

    area_1 overall:
      Speedup:     26,24 ms /   19,29 ms =    1,36
      Real-Time:   19,29 ms /   26,24 ms =  -26,5%
      CPU-Time:    38,58 ms /   26,24 ms =  +47,0%
```

Note that the decimal separator is dependent of the locale. This run was done on a German machine and therefore shows the German separator (comma). On an English machine the separator would be a point.

From the runfile report (second report) can see that threads are idle for about a third of the time overall. But only a little below 10% in the recorded areas. In the parallel area, the idle time is only 1.89%. The final comparison shows a speed-up of 1.30 in the recorded areas and 1.36 in the parallel area which in comparison with the idle time indicates a rather large overhead for the parallelization.

15 The HEAP-API C-library

The *HEAP-API* library provides the following functions. Unless noted otherwise, it is sufficient to include *heap_reporting.h*.

All library functions that report a thread ID have a sibling function that takes the thread ID as explicit argument. They have the form:

```
<normal\_function\_name>\_t( thread\_id, <normal\_function\_arguments> )
```

15.1 Management Functions

- `int heap_init( const char *output_file, int num_threads)`
  `int heap_report_init( const char *output_file, int num_threads )`

  Sets up the reporting library and opens the output file. Returns 0, if successful. `heap_init()` also initializes the parallelism facility of the *HEAP API* and sets `heap_mem_lock`, `heap_mem_unlock`, `heap_internal_lock`, `heap_internal_unlock` and `heap_get_thread_id` to the respective internal functions. `heap_init()` requires *HEAP_API.h*.

  - `output_file` – the path to the output file to be used (the runfile to be created). If set to *null*, *stdout* will be used and the data files will be named `run.data<number>.txt`.
  - `num_threads` – number of threads to be used (and initialized if `heap_init()` is used). Must be at least 1. This number must still be correct even if `heap_report_init()` is used: reporting a thread number to low, will result in problems with reporting. Reporting a number higher than actual, might result in bad performance values (as all higher threads appear to be idle at all times).

- `void heap_finalize()`
  `void heap_report_finalize()`

Closes the output file and cleans up all data structures, no calls to other functions are allowed afterwards. `heap_finalize()` also ends the threads created by `heap_init()` and closes open data ports. The finalization functions must correspond to initialization function: use `heap_finalize()` if `heap_init()` was used and use `heap_report_finalize()` if `heap_report_init()` was used. `heap_finalize()` requires *HEAP_API.h*.

- `void heap_report_enable( int enabled )`

  Enables/disables all reporting

  – `enabled` – 0 will disable all reporting, 1 (or any non-null value) will enable reporting. Default 1.

- `void heap_report_data_enable( int enabled )`

  Enables/disables reporting of input/output data.

  – `enabled` – 0 will disable data reporting, 1 (or any non-null value) will enable data reporting. Note that reporting must also be enabled – change via `heap_report_enable()`. Default 1.

15.2 Functions for Setting Up External Parallelism

- `void heap_set_get_thread_id_func( heap_get_thread_id_func get_thread_id_func)`

  Sets the function called when `heap_get_thread_id()` is used. Use this function to set a thread id function if a different parallelism facility is used.

  – `get_thread_id_func` – the function called to get the current thread's ID. Must suffice:
  `int ()()`
  Its return value must be a 0 based thread ID (i.e. 0, 1, 2, ...)

- `void heap_set_mem_lock_func( heap_lock_func mem_lock_func )`

  Sets the function called when `heap_mem_lock()` is called. Use this function to set a mem lock function if a different parallelism facility is used and thread-safety of `malloc()` and `free()` cannot be guaranteed.

  – `mem_lock_func` – the function called to lock the mutex before memory (de)allocation. Must suffice:
  `void ()()`

- `void heap_set_mem_unlock_func( heap_unlock_func mem_unlock_func )`

  Sets the function called when `heap_mem_unlock()` is called. Use this function to set a mem unlock function if a different parallelism facility is used and thread-safety of `malloc()` and `free()` cannot be guaranteed.

  - `mem_unlock_func` – the function called to unlock the mutex after memory (de)allocation. Must suffice:
    `void ()()`

- `void heap_set_internal_lock_func( heap_lock_func internal_lock_func )`

  Sets the function for locking the library's internal thread-safety lock. Use this function to set a mutex lock function if a different parallelism facility is used.

  - `internal_lock_func` – the function called to lock the mutex. Must suffice:
    `void ()()`

- `void heap_set_internal_unlock_func( heap_unlock_func internal_unlock_func )`

  Sets the function for unlocking the library's internal thread-safety lock. Use this function to set a mutex unlock function if a different parallelism facility is used.

  - `internal_unlock_func` – the function called to unlock the mutex. Must suffice:
    `void ()()`

15.3 Parallelism

- `void heap_add_task( heap_task_func task, void *arg )`

  Uses to create and add a task to the library's task management. Requires *HEAP_API.h*.

  - `task` – the function to call in order to execute the task. The function must suffice:
    `void ()( int thread_id, void *arg);`

    * `thread_id` – ID of the thread, 0 based.
    * `arg` – the argument as given by the call to `heap_add_task()`.

- `void heap_join()`

  Aids with task processing and returns once all tasks are completed. Should only be called by the main thread. Requires *HEAP_API.h*

- `int heap_get_thread_id()`

  Returns ID of current thread. Only works if either the *HEAP API* parallelism facilities are used (by calling `heap_init()` or the thread ID function has been set explicitly using `heap_set_get_thread_id_func()`.

- `void heap_mem_lock()`

  Locks the memory access mutex (recursive locking possible). Call this before allocating/deallocating memory if `malloc()` and `free()` are not thread-safe. Only works if either the *HEAP API* parallelism facilities are used (by calling `heap_init()` or the mem lock function has been set explicitly using `heap_set_mem_lock_func()`.

- `void heap_mem_unlock()`

  Unlocks the memory access mutex (recursive locking possible). Call this after allocating/deallocating memory if `malloc()` and `free()` are not thread-safe. Only works if either the *HEAP API* parallelism facilities are used (by calling `heap_init()` or the mem unlock function has been set explicitly using `heap_set_mem_unlock_func()`.

15.3.1 Data Ports

- `heap_port_handle heap_port_create( heap_port_id port_id, unsigned int data_size, int buffer_size )`

  Create a new data port with the given port ID, data size and buffer size. Returns the handle of the newly opened port.

  - `port_id` – a string with the port's ID
  - `data_size` – size of the data item the port is to receive in bytes. E.g. if the port is going to transport `int` data the size must be `sizeof(int)`
  - `buffer_size` – number of data items the port can store at anyone time

  Ports can be closed with `heap_port_close()` or automatically when the *HEAP API* library is finalized.

- `void heap_port_close( heap_port_handle handle )`

  Closes the port denoted by *handle*.

  - `handle` – the handle of the port that is to be closed

- `heap_port_handle heap_port_find( heap_port_id port_id )`

  Searches for an open port with the given ID and returns its handle or *HEAP_PORT_NONE* if no such port was found.

  - `port_id` – string with the ID of the port that is to be found

- `void heap_port_attach_stream( heap_port_handle port_handle, heap_stream_handle stream_handle, int auto_close_stream )`

  Attaches a report data stream to the data port for data reporting.

  - `port_handle` – handle of the port to which the data stream is going to be attached

  - `stream_handle` – handle of the (already opened) report data stream that is going to be attached to the data port. See documentation for `heap_report_data_stream()` for details on how to open a report data stream.
  - `auto_close_stream` – flag that indicates whether the report data stream should be closed when the data port is closed (0 = don't close, 1 = close)

- `heap_stream_handle heap_port_detach_stream( heap_port_handle port_handle, int close_stream )`

  Detaches a data stream from a data port and returns it. Returns the stream handle of the previously attached report data stream unless no such stream was attached or the stream was closed. The it returns *HEAP_STREAM_HANDLE_NULL*.

  - `port_handle` – handle of data port from which to detach the report data stream
  - `close_stream` – flag that indicates whether the detached report data stream should be closed (1 = close, 0 = don't close). If the stream was closed *HEAP_STREAM_HANDLE_NULL* will be returned

- `void heap_port_write_multiple( heap_port_handle handle, const void *data, int num_items )`
  `void heap_port_write( heap_port_handle handle, const void *data )`

  Writes data to a data port. `heap_port_write_multiple()` writes multiple data items to the stream, `heap_port_write()` only one. The function returns once the data has been written to the port. Note that this might include waiting for another thread to remove a sufficient amount of items from the port.

  - `handle` – handle of data port
  - `data` – pointer to data item /to array of data items
  - `num_items` – number of items to write to the data port. The buffer *data* points to must be contain at least that many items.

- `void heap_port_read_multiple( heap_port_handle handle, void *buffer, int items_to_read )`
  `void heap_port_read( heap_port_handle handle, void *buffer )`

  Reads data from a data port. `heap_port_read_multiple()` reads multiple data items from the stream, `heap_port_read()` only one. The function returns once the data has been read from the port. Note that this might include waiting for another thread to write a sufficient amount of items to the port.

  - `handle` – handle of data port
  - `buffer` – pointer to buffer to receive the item(s)
  - `items_to_read` – number of items to read from the data port. The buffer *buffer* points to must be large enough to contain at least that many items.

  Beware that the function will block until that many items have been written to the port. It is therefore necessary to make sure that these items have been written or will

be written some time in the future. Otherwise the function will block indefinitely. One can use `heap_port_data_available()` to verify that items are available.

- `unsigned int heap_port_data_size( heap_port_handle handle )`

  Returns the data size of a data port as specified during creation.

  – `handle` – handle of data port

- `int heap_port_buffer_size( heap_port_handle handle )`

  Returns the buffer size (in data items) of a data port as specified during creation.

  – `handle` – handle of data port

- `int heap_port_data_available( heap_port_handle handle )`

  Returns the amount of data items currently stored in a data port.

  – `handle` – handle of data port

- `int heap_port_space_available( heap_port_handle handle )`

  Returns the amount of free buffer size (in data items) of a data port.

  – `handle` – handle of data port

-

15.4 Utilities

- `double heap_time()`

  High precision timer, returns a timestamp in seconds. Only relative measurements are meaningful. The implementation depends on the platform.

- `heap_area_id heap_get_area_id()`

  Returns current area id as set by `heap_report_start_sequential()` or `heap_report_start_parallel()`.

- `int heap_get_state()`
  `int heap_get_state_t( int thread_id )`

  Returns state of the current thread or respectively, the thread given by `thread_id`.

- `heap_task_id heap_get_task_id()`
  `heap_task_id heap_get_task_id_t( int thread_id )`

  Returns the ID of the task the current thread (or respectively the thread given by `thread_id`) is working on, as set by `heap_report_start_task()` or *HEAP_TASK_UNDEFINED* if no task is currently processed.

15.5 Reporting

- `void heap_report_state( int state )`
  `void heap_report_state_t( int thread_id, int state )`

  Sets state of current or given thread (for performance analysis). Only needs to be used if the *HEAP API* parallelism facilities are not used, otherwise the state is set automatically.

  – `state` – the new state of the thread. One of `HEAP_STATE_UNDEFINED`, `HEAP_STATE_IDLE`, `HEAP_STATE_WORK`, `HEAP_STATE_SYNC`, `HEAP_STATE_TERMINATED`.
  – `thread_id` – the ID of the thread, the state of which is to be set

15.5.1 Areas and Tasks

- `void heap_report_start_sequential( heap_area_id area_id )`

  Reports start of a sequential area with ID `area_id`.

- `void heap_report_end_sequential( heap_area_id area_id )`

  Reports end of a sequential area with ID `area_id`. ID must match previous call to start area.

- `void heap_report_start_parallel( heap_area_id area_id )`

  Reports start of a parallel area with ID `area_id`.

- `void heap_report_end_parallel( heap_area_id area_id )`

  Reports end of a parallel area with ID `area_id`. ID must match previous call to start area.

- `void heap_report_start_task( heap_task_id task_id )`
  `void heap_report_start_task_t( int thread_id, heap_task_id task_id )`

  Reports start of a task in a thread (current or given). The thread ID must be supplied explicitly if external parallelism facilities are used and no valid thread ID function is set.

  – `task_id` – ID of the task which now starts. Should be unique.
  – `thread_id` – ID of thread which starts processing the task.

- `void heap_report_end_task( heap_task_id task_id )`
  `void heap_report_end_task_t( int thread_id, heap_task_id task_id )`

  Reports end of a task in a thread. Must match previous call to `heap_report_start_task()` for this thread.

15.5.2 Data

- `void heap_report_data( heap_area_id area_id, heap_data_id data_id, const void *data, int length )`

  Used for reporting block data. Should only be applied between areas in order to maintain timing.

  - `area_id` – ID of area to which the data belongs
  - `data_id` – ID of data set must be unique per area /task and pass
  - `data` – pointer to data to dump
  - `length` – size of data in bytes

  The data will be dumped to a file named *¡runfile_name¿.data¡continous_number¿.¡runfile_ext¿*.

- `heap_stream_handle heap_report_data_stream( heap_area_id area_id, heap_data_id data_id, int buffer_size )`

  Creates a report data stream for reporting stream data.

  - `area_id` – ID of area to which the data belongs
  - `data_id` – ID of data set must be unique per area /task and pass
  - `buffer_size` – size of buffer in bytes. Set to -1 for default buffer size.

  The data will be dumped to a file named *¡runfile_name¿.data¡continous_number¿.¡runfile_ext¿*.

- `void heap_report_close_stream( heap_stream_handle handle )`

  Closes the report data stream given by *handle*. Report data streams are also closed automatically when the *HEAP API* library is finalized or when the report data stream is detached from a data port (and the corresponding flag was set) or when an attached data port is closed (and the corresponding flag was set).

- `void heap_report_add_to_stream( heap_stream_handle handle, const void *data, int data_size )`
  `void heap_report_add_to_stream_t( int thread_id, heap_stream_handle handle, const void *data, int data_size)`

  Writes data to a report data stream.

  - `handle` – handle of report data stream
  - `data` – pointer to the data.
  - `data_size` – size of data in bytes.
  - `thread_id` – ID of thread that writes the data

  Data can also be written to a report data stream by attaching it to a data port and writing data to the data port.

- `void heap_report_flush_stream( heap_stream_handle handle )`
  `void heap_report_flush_stream_t( int thread_id, heap_stream_handle handle )`

  Flushes buffered data in a report data stream to disk.

  – `handle` – handle of report data stream
  – `thread_id` – ID of thread that writes the data

  Buffered data will automatically be flushed when the buffer is full or the stream is closed.

15.5.3 Assertions and Checkpoints

- `void heap_report_assert( int success, const char *assertion, const char *message )`
  `void heap_report_assert_t( int thread_id, int success, const char *assertion, const char *message )`
  `HEAP_ASSERT_MSG( assertion, message )`
  `HEAP_ASSERT( assertion )`
  `HEAP_ASSERT_MSG_T( thread_id, assertion, message )`
  `HEAP_ASSERT_T( thread_id, assertion )`

  Reports assertions and their result, also performs them if one of the macros is used. The thread ID must be supplied explicitly if external parallelism facilities are used and no valid thread ID function is set.

  – `success` – whether the assertion test was successful. The macro version determine that themselves by evaluating `assertion`.
  – `assertion` – the assertion's test
  – `message` – optional message to attach to assertion. May be *null*.
  – `thread_id` – ID of current thread

- `void heap_report_chkpnt_unordered( heap_chkpnt_id chkpnt_id )`
  `void heap_report_chkpnt_unordered_t( int thread_id, heap_chkpnt_id chkpnt_id )`

  Reports an unordered checkpoint. The thread ID must be supplied explicitly if external parallelism facilities are used and no valid thread ID function is set.

  – `chkpnt_id` – ID of checkpoint, should be unique.
  – `thread_id` – ID of current thread

- `void heap_report_chkpnt_ordered( heap_chkpnt_id chkpnt_id )`
  `void heap_report_chkpnt_ordered_t( int thread_id, heap_chkpnt_id chkpnt_id )`

Reports an ordered checkpoint. The thread ID must be supplied explicitly if external parallelism facilities are used and no valid thread ID function is set.

- `chkpnt_id` – ID of checkpoint, should be unique.
- `thread_id` – ID of current thread

16 Appendix

16.1 Quicksort Analysis Output

```
Analyzing runfile to check ('data/quick_sort_run.txt')
  verifying
Analyzing reference runfile ('data/quick_sort_seq_run.txt')
  verifying
Loading structure file ('data/quick_sort_structure.txt')
Verification of data/quick_sort_run.txt
  Comparing data
    comparing 'data/quick_sort_run.txt' and 'data/quick_sort_seq_run.txt'
    processing area area_0 pass 1
    processing area area_1 pass 1
    all data matches
  Checking order
    comparing 'data/quick_sort_run.txt' and 'data/quick_sort_seq_run.txt'
    processing area area_0 pass 1
    processing area area_1 pass 1
    all checkpoints match
  Checking assertions
  checking assertions in 'data/quick_sort_run.txt'
    checking ASSERTION: thread 0 @ 0.0538460000007035: error_found == -1
- final sort check
    all assertions encountered are ok
  all checks successful

all file checks completed successfully


Coverage analysis
  collecting checkpoint data
    from data/quick_sort_run.txt
  performing integrity checks
```

```
  comparing to structure file

  area coverage:
    coverage: 2 / 2 =  100,00%

  unordered checkpoint coverage:
    coverage: 1 / 1 =  100,00%

  ordered checkpoint coverage:
    ordered checkpoint not encountered: o7
    coverage: 6 / 7 =  85,71%

  total checkpoint coverage:
    coverage: 7 / 8 =  87,50%

Performance analysis of reference data/quick_sort_seq_run.txt
  compiling performance report for data/quick_sort_seq_run.txt
================================== Reference ============================
Performance report:
runfile data/quick_sort_seq_run.txt: start @    0,00 ms
  WORK      :   59,67 ms = 100,00%
  total     :   59,67 ms

  in areas only: start @    0,00 ms
    WORK      :   29,66 ms = 100,00%
    total     :   29,66 ms

  area area_0: start @    0,40 ms
    WORK      :    3,42 ms = 100,00%
    total     :    3,42 ms

  area area_1: start @   19,08 ms
    WORK      :   26,24 ms = 100,00%
    total     :   26,24 ms



Performance analysis of data/quick_sort_run.txt
  compiling performance report for data/quick_sort_run.txt
================================== data/quick_sort_run.txt ==============
Performance report:
runfile data/quick_sort_run.txt: start @    0,00 ms
    IDLE      :   35,37 ms =  32,80%
```

```
    WORK      :   72,45 ms =  67,20%
    total     :  107,82 ms

    in areas only: start @    0,00 ms
      IDLE      :    4,29 ms =   9,38%
      WORK      :   41,41 ms =  90,62%
      total     :   45,70 ms

    area area_0: start @    0,53 ms
      IDLE      :    3,56 ms =  50,00%
      WORK      :    3,56 ms =  50,00%
      total     :    7,11 ms

    area area_1: start @   19,80 ms
      IDLE      :    0,73 ms =   1,89%
      WORK      :   37,85 ms =  98,11%
      total     :   38,58 ms


---------------------------------- Comparison ----------------------------
Comparison report:
  overall:
    Speedup:     59,67 ms /   53,95 ms =    1,11
    Real-Time:   53,95 ms /   59,67 ms =   -9,6%
    CPU-Time:   107,82 ms /   59,67 ms =  +80,7%

  in areas only:
    Speedup:     29,66 ms /   22,85 ms =    1,30
    Real-Time:   22,85 ms /   29,66 ms =  -23,0%
    CPU-Time:    45,70 ms /   29,66 ms =  +54,1%

  per Area:
    area_0 overall:
      Speedup:      3,42 ms /    3,56 ms =    0,96
      Real-Time:    3,56 ms /    3,42 ms =   +4,0%
      CPU-Time:     7,11 ms /    3,42 ms = +108,0%

    area_1 overall:
      Speedup:     26,24 ms /   19,29 ms =    1,36
      Real-Time:   19,29 ms /   26,24 ms =  -26,5%
      CPU-Time:    38,58 ms /   26,24 ms =  +47,0%
```